Mancoosi Deliverable D5.4: Report on the international competition

Pietro Abate, Ralf Treinen

mancoosi

managing software complexity

Report on the international competition

Deliverable 5.4

Nature : Deliverable
Due date : 31.05.2011
Start date of project : 01.02.2008
Duration : 40 months

SEVENTH FRAMEWORK PROGRAMME

Information Society Technologies

Specific Targeted Research Project
Contract no.214898
Seventh Framework Programme: FP7-ICT-2007-1

A list of the authors and reviewers

Project acronym	Mancoosi
Project full title	Managing the Complexity of the Open Source Infrastructure
Project number	214898
Authors list	Pietro Abate Ralf Treinen
Workpackage number	WP5
Deliverable number	4
Document type	Deliverable
Version	1
Due date	31/05/2011
Actual submission date	19/05/2011
Distribution	Public
Project coordinator	Roberto Di Cosmo

Abstract

The main objective of Workpackage 5 was to organize an international competition of solvers of package upgradeability problems. This deliverable reports on the organisation of the MISC 2010 competition, the results and the lessons learned, and about the evolution since 2010.

Contents

Chapter 1

Introduction

The main objective of Workpackage 5 was to organize an international competition of solvers of package upgradeability problems. These problems are expressed in the CUDF format developed by the Mancoosi project [TZ08, TZ09a]. A problem instance expressed in CUDF contains the metadata of currently installed, as well as of available packages, and a request to install new packages, remove currently installed packages, or upgrade packages to newer versions. The metadata of a package is comprised, among others, of name and version of packages, dependency, conflicts and provided virtual packages.

Problems of this kind are theoretically NP-complete [DCMB+06]. However, if a problem instance is satisfiable then it typically has a huge number of solutions. One of the goals of the Mancoosi project was to design a language of optimization criteria that would allow a user to specify what would constitute the best among all the solutions.

Hence, we consider the package upgradeability problem as a combinatorial optimization problem. The goal of organizing an international competition of solvers for such problems was to make the research community aware of this problem, and to promote this particular instance of combinatorial optimization in the different research communities. The competition was designed to be open to any interested participant, including but not limited to, the solvers developed in Workpackage 4 of the project.

Chapter 2

Before MISC 2010

Organizing a solver competition is a complex task. Besides organizational issues, the technical setup of the competition proved to be challenging for the Mancoosi team. For this reason we decided to prepare the official international MISC competition by some trial runs (MISC-live 1 and MISC-live 2) with the goal of detecting early the problems in the setup, and thus avoiding them in the first official competition. The first trial run, called at that time *Mancoosi internal Solver Competition* (MiSC), was run in January 2010 followed by a second trial run before the official MISC competition in June 2010. Because of the positive feedback we got from the community after the first trial run, and because of the benefit to allow solver developers to continually test their solvers in a formal setting, we decided after MiSC to repeat such trial runs at irregular intervals in the future, and we baptized these trial runs *MISC-live*.

The purpose of running an internal competition was two-fold. On one hand it allowed the organizers of the competition to test their setup in a realistic setting. Since the MISC competition was the first of its kind the organizers had to experiment with the definition of the rules of the competition and the technical setup. Experience from the SAT community[1] proved very helpful in this regard. However, because of the inherent differences between the two competitions, we were unable to reuse their infrastructure and we were forced to build the execution and ranking environment from scratch.

On the other hand, these trial runs were most helpful to the prospective participants of the official competition. Because of the technical complexity of the CUDF format and its semantic, we felt that it was necessary to provide a strong support to test and run prospective solvers in a realistic test environment which is as close as possible to the one used in the official competition.

This internal competition was open to all interested participants, both internal to the Mancoosi project and external to the project. Participants of this internal competition were :

- **apt-pbo:** a modified apt tool using pseudo-Boolean-optimization (`minisat+` or `wbo`), submitted by Mancoosi partner Caixa Mágica.
- **p2cudf:** a solver built on top of the Eclipse Provisioning Platform p2, based on the SAT4J library and submitted by a team from Université d'Artois and IBM which is not part of the Mancoosi project.
- **inesc-udl:** a SAT-based solver using the p2cudf parser (from Eclipse) and the MaxSAT solver MSUnCore, submitted by Mancoosi partner Inesc-ID.

[1] `http://www.satlive.org/`

- **unsa:** a solver built using ILOG's CPLEX, submitted by Mancoosi partner Université Nice/Sophia-Antipolis.

The trial runs proved indeed very useful both for the participants and for the organizers. For instance, some of the participants found at MISC Live 1 that their solvers failed for some unexpected but trivial reason, like some mistake in the implementation of the parser for the input language. The most important lesson for the organizers of MISC Live 1 was that the initial system of limiting the resources of solver processes, based on the `ulimit` utility, was not sufficient since it did not allow to notify the solvers when they were about to overstep their time limit. As a consequence of this experience, the resource limitation was reimplemented from scratch, using the `runsolver` utility due to Olivier Roussel from Sat Live.

Chapter 3

MISC 2010

The first official international competition, the *Mancoosi International Solver Competition (MISC)*, was run in the summer of 2010. The competition was announced on various mailing lists and electronic newsletters. The timeline was as follows:

- April 22, 2010: official announcement of the competition.
- May 31, 2010: participants have to declare by email their intention to participate.
- June 13, 2010: deadline for the submission of solvers.
- July 10, 2010: announcement of the results.

The results of the competition were announced at the *First International Workshop on Logics for Component Configuration (LoCoCo 2010)*. LoCoCo 2010 took place during SAT 2010, the Thirteenth International Conference on Theory and Applications of Satisfiability Testing. Both LoCoCo 2010 and SAT 2010 were part of the Federated Logic Conference 2010, which was the major international event for computational logics in 2010 consisting of 8 main conferences and about 50 workshops. The FLoC conference is held every three to four years as a federation of most of the major international conferences on computational logics.

3.1 The Rules of MISC 2010

All problems used in the competition are expressed in the CUDF format developed by the Mancoosi project [TZ08, TZ09a]. A CUDF document contains a complete abstraction of a software installation on a user machine plus the user request. It contains the metadata of currently installed, as well as of available packages, and a request to install new packages, remove currently installed packages, or upgrade packages to newer version. The metadata of a package is comprised, among others, of name and version of packages, dependency, conflicts and provided virtual packages. An overview of the CUDF format is given below in Subsection 3.1.1.

Since, if a solution exists, there exist in general many of them, one also needs a criterion to distinguish an optimal solution (which is not necessarily unique). We hence proposed a system to define precisely different complex optimization criteria. The exact definition of this system is given below in Subsection 3.1.2. For MISC 2010 we had two different tracks, each of them defined by a different optimization criterion

3.1.1 Problem Format

CUDF (for Common Upgradeability Description Format) is a specialized language used to describe upgrade problem instances, that is presented in detail in [TZ08]. Here we just recall that it has been designed with the following goals in mind:

Platform independence We want a *common* format to describe upgrade scenarios coming from diverse environments. As a consequence, CUDF makes no assumptions on a specific component model, version schema, dependency formalism, or package manager.

Solver independence In contrast to encodings of inter-component relations which are targeted at specific solver techniques, CUDF stays close to the original problem, in order to preserve its structure and avoid bias towards specific solvers.

Readability CUDF is a compact plain text format which makes it easy for humans to read upgrade scenarios, and facilitates interoperability with package managers.

Extensibility Only core component properties that are shared by mainstream platforms and essential to the meaning of upgrade scenarios are predefined in CUDF. Other auxiliary properties can be declared and used in CUDF documents, to allow the preservation of relevant information that can then be used in optimization criteria, e.g. component size, number of bugs, etc.

Formal semantics CUDF comes with a rigorous semantics that allows package managers and solver developers to agree on the meaning of upgrade scenarios. For example, the fact that self-conflicts are ignored is not a tacit convention implemented by some obscure line of code, but a property of the formal semantics.

Figure 3.1 shows an excerpt of a CUDF document in a sample upgrade scenario. A CUDF document consists in general of three parts:

- A preamble, which in particular may declare additional package properties, together with their type, that may be used in the package metadata. In the example of Figure 3.1 we have declared an additional package property called `bugs` of type *int*, this property is optional and the default value is 0.
- A list of package stanzas, each of them started on the keyword `package`. This list gives the metadata of all the packages that are currently installed on the system, or available for future installation.
- A request stanza that defines the problems to be solved. In the example, we require of a solution that it must contain (have installed) some package called `bicycle`, and also some package called `gasoline-engine` in version 1.

A package stanza is a record consisting of some standard fields, plus additional fields if declared in the preamble. The most important package fields are package name and version (which together serve to uniquely identify a package), dependencies and conflicts. The `dependency` field of a package p specifies other packages that must be installed together with the given

```
preamble:
property: suite: enum(stable,unstable) = "stable"
property: bugs: int = 0

package: car
version: 1
depends: engine, wheel > 4 , door < 7, battery
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine
installed: true

...

request:
install: bicycle, gasoline-engine = 1
```

Figure 3.1: Sample CUDF document.

package p, and the `conflicts` field is exclusion list of packages that must not be installed together with p. Dependencies are conjunctions of disjunctions, the elements of which are package names, probably together with a version constraint.

3.1.2 Optimization Criteria

Two different fixed optimization criteria were used for MISC 2010, each of them defining one track in the competition. Both criteria are lexicographic combinations of different integer-valued utility functions to be evaluated w.r.t. a given solution.

Each of these utility functions calculates a value only from the initial installation status and a given solution. These utility function have therefore a global scope. As a consequence, the user request is not included the optimization criterion itself. The reason for this is that each proposed solution must first of all satisfy the user request, and only if this is the case is compared with other solutions w.r.t. the given optimization criteria. Solutions that do not satisfy the user request will be discarded and the solver will obtain a penalty value. Only those of the proposed solutions that actually satisfy the user request will yield a score according to the ranking function described below in Subsection 3.1.3.

Two tracks were proposed for MISC: the *paranoid* and the *trendy* track. The motivation behind choosing these two criteria was to model the intentions of two different categories of users: the *sysadm* - paranoid track - which privileges solutions that change the least number of packages, and the *gamer* - trendy track - which prefers the latest version of software installed on the system, and who does not strongly oppose to installing additional packages on the system.

We have choosen two fixed criteria for the competition, in contrast to a domain-specific language allowing for the free definition of functions like MooML [TZ09b], in order to not overburden participants of the competition with a too complex input language. For the next edition of MISC, however, we have generalized the choice of optimization criteria (see Section 4).

The first criterion ("paranoid") uses only standard package properties as defined in the CUDF standard. For the second criterion ("trendy") we made use of an extra package property which is not among the mandatory package properties. This additional property is `recommends`, that is expressed as a conjunctions of disjunctions of package, possibly with version constraints (as the property `depends`).

The idea is that recommended packages are non-mandatory dependencies. For that reason, the CUDF documents used as problem instances in the competition contain a specification of that property in the document preamble, like this:

```
preamble:
property: recommends: vpkgformula = [ true! ]
```

The precise definition of the different utility functions used in the formal definitions of the criteria are defined as follows. Suppose that a solver tool outputs for the initial universe I (the set of initially installed packages) a proposed solution S. We write $V(X, name)$ for the set of versions in which *name* (the name of a package) is installed in X, where X may be I or S. That set may be empty (if *name* is not installed), contain one element (if *name* is installed in exactly that version), or even contain multiple elements in case a package is installed in multiple versions.

We write

- $\#removed(I, S)$ is the number of packages removed in the proposed solution S w.r.t. the original installation I:

$$\#removed(I, S) = \#\{name \mid V(I, name) \neq \emptyset, V(S, name) = \emptyset\}$$

- $\#new(I, S)$ is the number of new packages in the proposed solution S w.r.t. the original installation I:

$$\#new(I, S) = \#\{name \mid V(I, name) = \emptyset, V(S, name) \neq \emptyset\}$$

- $\#changed(I, S)$ is the number of packages with a modified (set of) version(s) in the proposed solution S w.r.t. the original installation I:

$$\#changed(I, S) = \#\{name \mid V(I, name) \neq V(S, name)\}$$

- For the next definition we need the notion of *latest* version of package: We write $latest(name)$ for the most recent available version of a package with name *name*. We define $\#notuptodate(I, S)$ as the number of installed packages but not in the latest available version:

$$\#notuptodate(I, S) = \{name \mid V(S, name) \neq \emptyset, latest(name) \notin S\}$$

- $\#unsatisfied - recommends(I, S)$ counts the number of disjunctions in Recommends-fields of installed packages that are not satisfied by S:

$$\#unsatisfied - recommends(I, S) = \{(name, v, c) \mid v \in V(S, name), c \in recommends(name, v), S \not\models c\}$$

In that definition, $recommends(name, v)$ denotes the set of recommendations of the package with name $name$ and version v. Each such recommendation may be a single package, possible with a version constraint, or in general a disjunction of these. Such a recommendation c is satisfied by a proposed installation S, written $S \models c$, if at least one of the elements of the disjunction c is realized by S.

For instance, if package a recommends

$$b,\ c \mid d \mid e,\ e \mid f \mid g,\ b \mid g,\ h$$

and if S installs a, e, f, and h, but neither of b, c, d, or g , then one would obtain for the package a alone a value of 3 for #unsatisfied-recommends since the 2nd, 3rd and 5th disjunct of the recommendation are satisfied, and the others are not. If no other package contains recommendations then #unsatisfied-recommends(I,S)=3. Note that in any case the value of #unsatisfied-recommends(I,S) only depends on S but not on I.

The two optimization criteria are now defined as follows:

- *paranoid*: we want to answer the user request, minimizing the number of packages removed in the solution, and also the packages changed by the solution. Formally:

  $$lex(min\#removed, min\#changed)$$

  Hence, two solutions S_1 and S_2 will be compared as follows:

  1. compute, for $i \in \{1, 2\}$:

     $$\begin{aligned} r_i &= \#removed(I, S_i) \\ c_i &= \#changed(I, S_i) \end{aligned}$$

  2. S_1 is better than S_2 iff $r_1 < r_2$ or ($r_1 = r_2$ and $c_1 < c_2$).

- *trendy*: we want to answer the user request, minimizing the number of packages removed in the solution, minimizing the number of outdated packages in the solution, minimizing the number of unsatisfied recommendations, and minimizing the number of extra packages installed. Formally:

  $$lex(min\#removed, min\#notuptodate, min\#unsatisfied - recommends, min\#new)$$

  Hence, two solutions S_1 and S_2 will be compared as follows:

  1. compute, for $i \in \{1, 2\}$:

     $$\begin{aligned} r_i &= \#removed(I, S_i) \\ u_i &= \#notuptodate(I, S_i) \\ ur_i &= \#unsatisfied - recommends(I, S_i) \\ n_i &= \#new(U, S_i) \end{aligned}$$

  2. S_1 is better than S_2 iff $r_1 < r_2$ or ($r_1 = r_2$ and ($u_1 < u_2$ or ($u_1 = u_2$ and ($ur_1 < ur_2$ or ($ur_1 = ur_2$ and $n_1 < n_2$))))).

3.1.3 Ranking of Solutions

All participating solvers were executed on the same set of input problems. The ranking routine operates in three steps. First, all individual problems are ranked based on the solvers' output. Second, all individual solutions are numerically ranked, aggregating problems by track and problem set. In the end, the aggregate solutions for each track are aggregated together to determine the winner.

We first define how several participating solvers are ranked on one particular problem instance.

Let m be the number of participating solvers. For each input problem we give points to each solver, according to whether the participant yields

1. a claimed solution that really is a solution
2. FAIL (that is, the solver declares that he has not found a solution)
3. ABORT (timeout, segmentation fault, ...)
4. a claimed solution that in reality is not a solution to the problem

Note: After MIC 2010 we found that the difference between FAIL and ABORT is somewhat artificial. A *smart* solver might want to respond FAIL whenever it was able to detect a crash or a timeout. Despite the fact that this might prove the solver unsound, it will be ranked better then a solver that decided not to declare ABORT as FAIL. For this reason, in further editions of MISC (see Section 5), the cases FAIL and ABORT will be ranked the same.

The number of points a solver gets on a given input problem is calculated as follows:

1. All solvers in case (1) are ordered according to the optimization criterion (best solution first), and a participant in case (1) gets the number of points that corresponds to his position in that list. In case two solvers have found equally good solutions they get the same number of points. In other words, the number of points obtained be a solver is 1 plus the number of solvers that are strictly better.

   To give an example with 4 participating solvers s_1, s_2, s_3 and s_4: If s_1 gives the best solution, followed by s_2 and s_3 giving an equally good solution, and s_4 who gives the worst solution, then s_1 gets 1 point, s_2 and s_3 each get 2 points, and s_4 get 4 points.
2. A participant in case (2) gets $2 * m$ points.
3. A participant in case (3) gets $3 * m$ points.
4. A participant in case (4) gets $4 * m$ points

Note that it is possible that an input problem indeed doesn't have a solution. In that case, (1) above is not possible, so the solvers that correctly say FAIL are ranked best.

The total score of a solver is the sum of the number of points obtained for all problems of the problem set. The solver with the minimal score wins.

If several solvers obtain the best total score then the winner is determined by choosing among those having obtained the best total score the one with the minimal total success time. The total success time of a solver is the sum of execution times of that solvers over all problems in

case 1 (correct solution), plus $f * timeout$ where f is the number of problems where the result of that solver is in one of the cases 2 (FAIL) or 3 (abort, timeout,...) or 4 (wrong solution), and *timeout* is the time limit defined in the execution environment.

3.2 The Execution Environment of MISC 2010

The solver is executed on a virtual machine that simulates a GNU/Linux host of architecture x86, 32 bit, single-processor. On the virtual machine are installed:

- A standard POSIX environment. In particular a POSIX-compatible shell, which may be invoked by /bin/sh, is available, as well as a GNU Bourne-Again SHell (invoked as /bin/bash).
- A Java Runtime Environment (JRE) 6; the Java application launcher can be invoked as /usr/bin/java.
- A Python 2.5 environment; the main Python interpreter can be invoked as /usr/bin/python.

No other libraries, for any programming language, can a priori be assumed to be available in the virtual machine environment. This means:

- Solvers prepared using compilers supporting compilation to ELF must be submitted as statically linked ELF binaries. No assumptions can be made on available shared libraries (.so) in the virtual machine environment.
- Solvers written in some (supported) interpreted language must use the usual shebang lines (#!/path/to/interpreter) to invoke their interpreter.
- Solvers written in Java must be wrapped by shell scripts invoking the Java application launcher as needed.
- Solvers needing other kind of support in the virtual machine must make specific arrangements with the competition organizers before the final submission.

The solver execution is additionally constrained by the following limits (as implemented by ulimit):

- Maximum execution time: 5 minutes
- Maximum memory (RAM): 1 GB
- Maximum file system space: 1 GB (including the solver itself)

The execution time is CPU time, measured in the same way as `ulimit -t`. About 20 seconds before the timeout, the signal SIGUSR1 will be sent to the solver process. It is the responsibility of the solver to catch that signal, produce an output, and terminate normally. The resource limitation and the notification of the solvers is implemented in the execution environment by using the `runsolver` utility due to Olivier Roussel from Sat Live.

3.3 Requirements on Participating Solvers

The following instructions on how solvers are to be invoked were given to the participants:

- The solver will be executed from within the solver directory.
- Each problem of the competition will be run in a fresh copy of the solver directory.
- The solver will be called with 2 command line arguments: *cudfin* and *cudfout*, in this order. Both arguments are absolute file systems paths.
  1. *cudfin* points to a complete CUDF 2.0 document, describing an upgrade scenario. The CUDF document is encoded as ASCII plain text, not compressed.

     According to the specifications, the document consists of an optional preamble stanza, followed by several package stanzas, and is terminated by a request stanza.
  2. *cudfout* points to a non-existing file that should be created by the solver to store its result.

Solvers were expected to yield their solution as follows:

- The solver's standard output may be used to emit informative messages about what the solver is doing.
- If the solver is able to find a solution, then it must:
  - write to *cudfout* the solution encoded in CUDF syntax as described in Appendix B of the CUDF 2.0 specification;
  - exit with an exit code of 0.
- If the solver is unable to find a solution, then it must:
  - write to *cudfout* the string "FAIL" (without quotes), possibly followed by an explanation in subsequent lines (lines are separated by newline characters, ASCII 0x0A);
  - exit with an exit code of 0.
- All exit codes other than 0 will be considered as indications of unexpected and exceptional failures not related to the inability to find a solution.

3.4 Selection of Input roblems

The corpus of input problems selected for the MISC competition consists of two kinds of problems:

- problems originating from real world scenarios that users have encountered, or might encounter, during the normal evolution of a software distribution,
- purely artificial problems, with the goal to foster the advancement of combinatorial solving techniques at hand of challenging problems.

In the first class of problems originating from real-world scenarios, the organizers selected a number of CUDF documents distilled from the community-submitted DUDF upgrade problem reports [AGL+10] that were submitted to the central repository at UPD [AT11], namely from the Debian, Mandriva and Caixa Mágica communities.

Other then these submitted problems, we generated a number of artificial problems. The utility used is called `randcudf` and it was freely available to all participants to train their solvers. Despite the name of the utility, the generated problems are not purely random CUDF instances, but they are based on a common real universe. From this starting point, `randcudf` is able to generate random upgrade requests and to disturb the initial installation status according to a number of configuration parameters. The rationale behind this choice was to provide random variations of real problems that were increasingly challenging.

Among the parameters of randcudf are:

- the size of the universe: a universe can be composed by the union of all packages coming from different official releases. This allowed us to simulate the - usual - case where users mix packages coming from old and new software repositories.

- a list of packages declared as installed in the universe (status). We can associate to the same universe different installation status effectively changing the nature of the problem.

- the probability of generating install / remove requests with a version constraint. This parameter controls the CUDF request stanza and permits to adjust the constraints of individual requests.

- the number of packages declared as installed but whose dependencies might not be satisfied. This parameter allows to disturb the given base installation status effectively creating a completely different problem.

- the number of packages marked as `keep` and the type of keep (version or package). A package declared as `keep` in CUDF adds an additional constraint to be respected by the solver.

The combination of these variables allowed us to to produce problems of different size and different degree of complexity. For the official competition we generated three problem categories, with, respectively, a universe size with 30.000 (called *easy*), 50.000 (called *difficult*) and 100.000 packages (called *impossible*).

Moreover, to further control the type of proposed problems, we discarded all problems for which it is impossible to find a solution.

One last category of artificial problems we used in the competition were obtained by encoding into CUDF well-known difficult instances of 1-IN-3 SAT, a problem which is known to be NP-complet [Sch78].

Finally, the following five categories of problems instances were used in both tracks of the MISC competition :

Name	Description		
cudf_set	Encoding in cudf of 1-in-3-SAT		
debian-dudf	Submitted installation problems		
Name	Base Universe	Status	RandCudf Parameters
easy	Debian unstable	desktop installation from unstable	10 install, 10 remove
difficult	Debian stable, unstable	server installation from stable	10 install, 10 remove, 1 upgrade all
impossible	Debian oldstable, stable, testing, unstable	server installation from oldstable	10 install, 10 remove, 1 upgrade all

Table 3.1: MISC 2010 Problem Selection

3.5 Results

3.5.1 The Participants of MISC 2010

The following solvers were participating in MISC 2010:

- **apt-pbo**: apt-get replacement using a PBO solver, submitted by Mancoosi partner Caixa Mágica.
- **aspcud**: a CUDF-Solver based on Answer Set Programming using Potassco, the Potsdam Answer Set Solving Collection, submitted by a team from University of Potsdam.
- **inesc**: a SAT-based solver using the p2cudf parser (from Eclipse) and the MaxSAT solver MSUnCore, submitted by Mancoosi partner Inesc-ID.
- **p2cudf**: a family of solvers on top of the Eclipse Provisioning Platform p2, based on the SAT4J library, submitted by a team from Université d'Artois and IBM.
- **ucl**: a solver based on graph constraints, submitted by Mancoosi partner Université Catholique de Louvain,
- **unsa**: a solver built using ILOG's CPLEX, submitted by Mancoosi partner Université Nice/Sophia-Antipolis

3.5.2 Outcome

The results of the MISC competition highlight the unsa solver from the Université Nice/Sophia-Antipolis clearly as the overall winner in both categories, and at the same time show that a clever encoding tailored for a specific category can still win over a general purpose encoding. In particular, the solver p2cudf is the clear winner of the `debian-dudf` category, providing high quality results.

The two tables below summarize the scores obtained by the participants in both categories. The execution time was measured for each solver execution but is not shown in the tables since it was not necessary to break any ties. In general, on easy instances, all solvers took between 3 and 30 seconds to complete, while on difficult instances, the entire allotted time was used (300 secs).

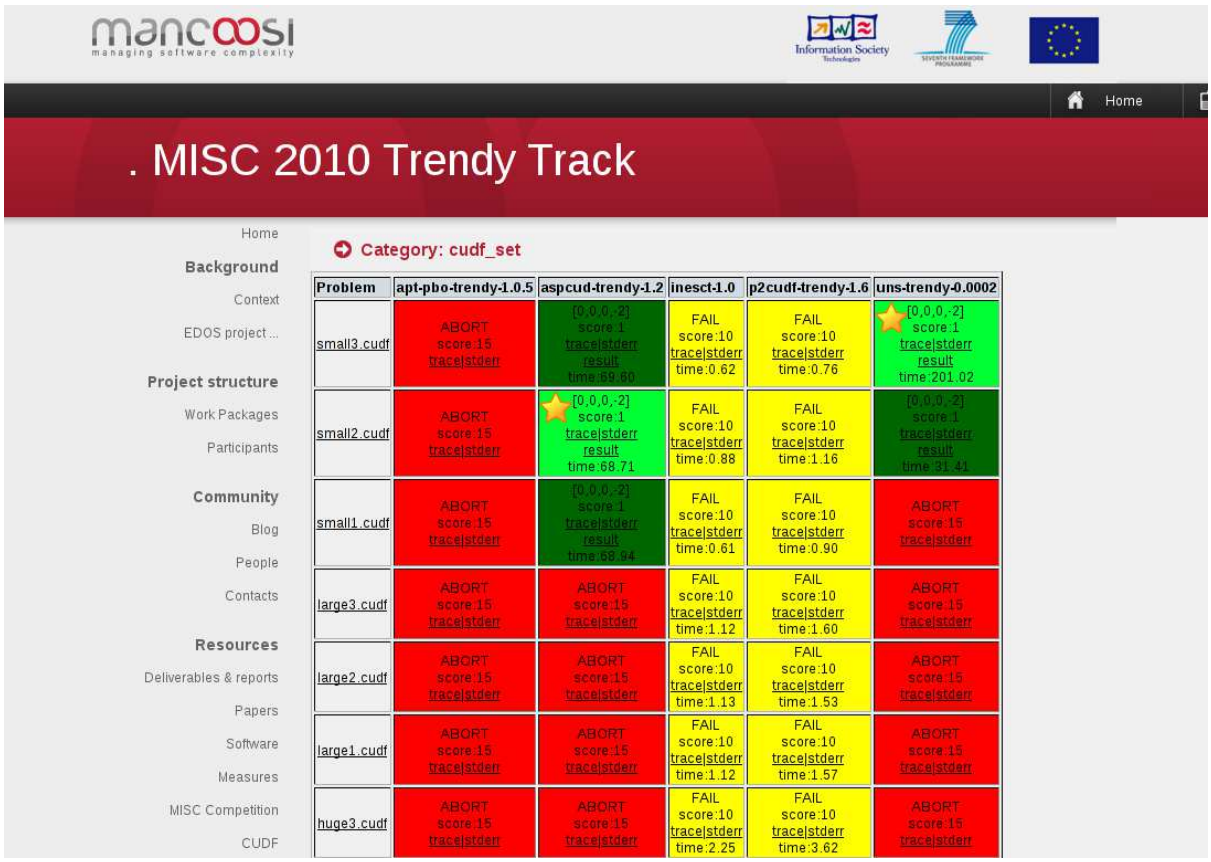
Category	apt-pbo	aspcud	inesc	p2cudf	uns
cudf-set	135	93	**90**	90	107
debian-dudf	211	189	194	**39**	74
difficult	415	70	101	98	**49**
easy	410	46	64	61	**21**
impossible	225	190	220	181	**15**
Total	1396	588	669	469	**266**

Table 3.2: Trendy Track Summary

Category	apt-pbo	aspcud	inesc	p2cudf	ucl-cprel	uns
cudf-set	162	118	108	108	108	**111**
debian-dudf	236	222	32	**32**	216	86
difficult	522	58	92	99	264	**55**
easy	504	21	63	63	252	**21**
impossible	300	270	120	120	180	**15**
Total	1724	689	415	422	1020	**288**

Table 3.3: Paranoid Track Summary

In addition to this summary, we also published a detailed table of results for each category, allowing all participants to inspect the solver trace (including `stderr` and `stdout`), the produced solution and the input problem. This information was meant to allow all participant to reproduce the execution and to further debug their solvers. A screen shot from the competition website is provided below:



The complete detailed results of the competition can be found at `http://www.mancoosi.org/misc-2010/results/paranoid/` for the paranoid track, and `http://www.mancoosi.org/misc-2010/results/trendy/` for the trendy track.

Chapter 4

Since MISC 2010

4.1 MISC-live 3 and 4

After having organized the MISC competition 2010, two more MISC-live competitions were organized in November 2010 (MISC-live 3) and then in February 2011 (MISC-live 4). These trial runs were used in particular to test the new third user track which will be used in the international competition 2011. The novelty of the user track is that one searches for an optimal solution according to an optimization criterion provided by the user. The criterion is given by a list of utility functions taken from a fixed list of possible functions, together with a polarity ("+" for maximize, or "-" for minimize) for each of them. This is a generalization of the first two tracks "paranoid" end "trendy", since they can now be seen as special instance of the new "user" track:

- the paranoid criterion could be written as

$$-\text{removed}, -\text{changed}$$

- the trendy criterion could be written as

$$-\text{removed}, -\text{notuptodate}, -\text{unmet\_recommends}, -\text{new}$$

4.2 Towards MISC 2011

The international competition MISC 2011 will be organized at the LoCoCo 2011 workshop, which will take place on September 12, 2011, in Perugia, Italy, as a satellite event of the 17th International Conference on Principles and Practice of Constraint Programming. The preliminary timeline is:

- Participants have to declare their intention to participate by August, 1.
- Deadline for submission for solvers is August, 12.

Chapter 5

Conclusion

Running a competition is not as easy as it may seem. Some of the difficulties we encountered are:

- One has to define a standard environment in which the submitted solvers are executed. The specification of the environment must be precise enough to prevent that s submitted solver fails for some trivial reason like non-availability of a library, different version as expected of a programming language systems or interpreters like perl, python or java, or too restrictive access rights to directories.
- One has to define precisely the interface of the participating solvers. This does not only include the syntax and semantics of the command-line arguments used in the competition to invoke the solver, but also specifying in which way resource restrictions on memory and cpu time are implemented. For instance, one participant of MISC live 3 was expecting that only the top-level process of the process group of his solver instance would receive a SIGUSR1 signal shortly before the timeout, where in reality the signal was sent to all processes. This unexpected signal received by a child process caused the solver to crash, instead of writing the best solution he had found up to that moment.
- Defining the right rules for electing the winner is tricky. For instance, we had in the MISC 2010 competition four different cases of the outcome of a solver on a particular problem instance: a solver could (1) give a correct result, (2) declare that he did not find a result, (3) crash, or (4) propose a solution which is wrong (i.e., a solution which is incoherent or does not satisfy the request). Only when reviewing the results of MISC 2010 we found that the distinction between cases (2) and (3) does not make sense since the difference between the two cases is just catching, or failing to do so, all exceptions that may be thrown.

  Another delicate question was how the running time of solvers should be taken into consideration. The problem is that on the one hand the running time is part of the user experience and should have some importance when ranking the participants, but that on the other hand a fast solver that yields bad solutions should never be able to win over a slow solver that finds very good solutions. We came up with a definition where running time is only taken into account for breaking ties between participants that have found equally good solutions in the average.
- Last but not least the results must be reproducible. This implies publishing all the problem instances used in the competition, instructions to rebuild the execution environment, and

source code of all the tools used for verifying and ranking solutions.

We are quite satisfied with the field of participants in the MISC and MISC-live competitions. In total, we have seen participating solvers from 4 different sites of the Mancoosi project, and 3 different teams who are not part of Mancoosi. Participating solvers are using quite different combinatorial solving techniques like SAT solving, mixed integer linear programming, pseudo-Boolean constraints, answer set programming, or graph constraints. Most of the participating solvers are totally or partly open source (the case where a solver is only partially open source usually happens when the frontend for encoding CUDF problems is open source but then uses a solving engine which itself is not open source).

Detailed information about the MISC-live trial runs, the first international MISC competition 2010, and upcoming MISC competitions can be found at the competition web site at `http://www.mancoosi.org/misc/`.

Bibliography

[AGL+10] Pietro Abate, André Guerreiro, Stéphane Laurière, Ralf Treinen, and Stefano Zacchiroli. Extension of an existing package manager to produce traces of ugradeability problems in CUDF format. Deliverable 5.2, The Mancoosi Project, August 2010. `http://www.mancoosi.org/reports/d5.2.pdf`.

[AT11] Pietro Abate and Ralf Treinen. UPDB infrastructure to collect traces of ugradeability problems in CUDF format. Deliverable 5.3, The Mancoosi Project, February 2011. `http://www.mancoosi.org/reports/d5.3.pdf`.

[DCMB+06] Roberto Di Cosmo, Fabio Mancinelli, Jaap Boender, Jerome Vouillon, Berke Durak, Xavier Leroy, David Pinheiro, Paulo Trezentos, Mario Morgado, Tova Milo, Tal Zur, Rafael Suarez, Marc Lijour, and Ralf Treinen. Report on formal mangement of software dependencies. Technical report, EDOS, 2006.

[Sch78] Thomas J. Schaefer. The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing, 1-3 May 1978, San Diego, California, USA*, pages 216–226. ACM, 1978.

[TZ08] Ralf Treinen and Stefano Zacchiroli. Description of the CUDF format. Deliverable 5.1, The Mancoosi Project, November 2008. `http://www.mancoosi.org/reports/d5.1.pdf`.

[TZ09a] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project, November 2009. `http://www.mancoosi.org/reports/tr3.pdf`.

[TZ09b] Ralf Treinen and Stefano Zacchiroli. Expressing advanced user preferences in component installation. In Roberto Di Cosmo and Paola Inverardi, editors, *IWOCE '09: Proceedings of the 1st international workshop on Open component ecosystems*, pages 31–40. ACM, August 2009.